An Implementation of Bounded Obligations

Martin S. Feather *
USC / Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292, USA
Email: feather@isi.edu

Abstract

An implementation of bounded obligations is demonstrated. Bounded obligations facilitate the expression of requirements such as 'a user must return a book on or before its due date'. *Our implementation translates a simple, declarative form of a bounded obligation into the equivalent data structures and operations necessary to ensure its adherence. Incremental development and exploration is achieved by modifying a 'strict' bounded obligation to become 'violatable' together with a way of recognizing when it has been violated. Again, a simple declaration to this effect is appropriately translated by our implementation. Finally, response to such violations can also be declared and automatically translated into running code. Our implementation is described and illustrated on examples drawn from a simple library system.*

1 Introduction

Many systems contain requirements in the form of *'bounded obligations'*, such as *'a user must return a book on or before its due date'*. It would be useful, therefore, to be able to state such obligations directly, and be able to reason about and execute prototypes and specifications containing them. In [12], a deontic action logic is used to specify the semantics of bounded obligations and of the error recovery associated with violations of those obligations. Herein we present an *implementation* of these same capabilities, which we think has value for the following reasons:

- The execution of a prototype/specification that makes use of bounded obligations is enabled, and thus the effects of the bounded obligations on behavior can be observed directly.
- Bounded obligations can be introduced *incrementally*, so that the specification can first make use of 'strict' obligations (that must always be met), then can be relaxed to allow violations of them, and finally augmented to respond to those violations. Each such version of the specification can be executed.
- Bounded obligations can be viewed as a special case of more general 'constraint programming'. The relative ease of translating bounded obligations into a language base of constraint programming facilities demonstrates the power of that base.
- More speculatively, bounded obligations are but one of a class of canonical exceptions — ways in which the 'normal', 'ideal' behavior of a system is gradually extended to accommodate deviations from that norm/ideal.

The remainder of this paper is organized as follows: in section 2 we describe bounded obligations and their relationship to software development; in section 3 we describe the language base into which our implementation translates bounded obligations; in section 4 we introduce the example to serve as illustration; in section 5 we describe our implementation that translates declarations of bounded obligations into the executable language base; finally, in sections 6 we discuss related work and conclusions.

2 Bounded Obligations and Software Development

Obligations — for example, that having opened a file, a system component must eventually close it — occur frequently in statements of system requirements and specifications. They dictate how the system and its environment should respond to one another, and

*Support from this work has been provided by Defense Advanced Research Projects Agency contract No. BAPT 63-91-K-0006; views and conclusions in this document are those of the author and should not be interpreted as representing the official opinion or policy of DARPA, the U.S. Government, or any other person or agency connected with them.

what to do when they fail to meet those requirements. Indeed, [15] argued cogently for their use as part of the description of module interfaces, and [13] showed a wide variety of forms of, and uses of, obligations. Often, obligations are *bounded*, that is, some stricter condition than 'eventually' is required.

Note that a *bounded obligation* is just a special form of temporal constraint. It involves two predicates, P and Q say; once the obligation has been established, P must become true *at or before* Q becomes true. For example, consider the following requirement (in the context of a simple lending library): *a user must return a book on or before its due date.* This is a bounded obligation whose predicate P is that the book be back in the library, and predicate Q is that the current date exceeds the book's due date. If the user returns the book on or before the due date, then the obligation has been met; if the due date passes but the user still hasn't returned the book, then the obligation has been violated.

A further property often desired of bounded obligations is that they be retractable — that is, having established such an obligation, it should be possible to retract the obligation before it becomes due. For example, if the borrower of a piece of rental equipment chooses to purchase that piece of equipment, payment of the purchase price should retract the obligation to return that rented equipment by its due date.

We believe it would be useful to be able to declare bounded obligations directly, rather then force the coding of their effects in terms of lower-level temporal primitives. We interpret this to be one of the points of [12], wherein the semantics of bounded obligations are given in terms of deontic logic, and once so described, used to specify a system in a clear and straightforward manner. This is, of course, in keeping with the focus of many specification languages, whose very purpose is to facilitate the direct expression of intent, requiring a minimum of manual translation on the part of the specifier to convert mental intent into equivalent formalism.

Another commonly shared belief is in the virtue of developing software *incrementally.* For example in developing the software for some system, we might begin with a simple form of the system in which only 'normal' behaviors are dealt with; once we are happy with the system in this form we then go on to consider what 'exceptional' cases might arise, and how to respond to them. It is important that bounded obligations should admit to this incremental development approach. In particular, we want to be able to start with *strict* obligations, which *must* be met (in specification terms, a specification containing such obligations would denote only behaviors that meet all of their obligations; in operational terms, if this is a running program, failure to meet an obligation would be manifested as a run-time error). Having dealt to our satisfaction with the system employing strict obligations, we would then 'relax' chosen obligations to allow exceptions to them to occur. Typically, we will then go on to introduce appropriate responses to such unmet obligations.

In summary, the claims upon which this paper rests are that:

- bounded obligations are useful for software specification, because they correspond to recurring forms of system requirements,
- to be widely useful, bounded obligations should be retractable, and/or violatable in such a way that observation of such violations can used to direct further processing, and
- incremental development of software may nee to proceed through stages from 'strict' obligations (which admit no violations) through violatable obligations, and accompanying responses to those violations.

In [12], deontic action logic was used to *specify the semantics* of bounded obligations meeting these desired qualities. The contribution of this paper is an elegant *implementation* of bounded obligations also exhibiting these desired qualities.

3 Target of Implementation, the Language AP5

In order to explain our implementation of extended obligations, we first describe the features of the target language into which we translate them.

This language is AP5 [4], our group's in-house relational database extension of Common Lisp. AP5 provides *relations* (of arbitrary arity) as the basic data representation for specification. The values of AP5's relations can be thought of as residing in an in-core database, against which the running AP5 program issues queries, and executes transactions to make changes to those values. The features of AP5 that we make most use of are those concerning the definition of relations, and the mechanisms augmenting database transactions. Additionally, AP5 and its associated compiler supports the implementation of relations in terms of more efficient storage structures, but since this aspect is not pertinent to our implementation of obligations, we will not discuss it further.

The key AP5 features that we do use are as follows:

Relations — typed relations of arbitrary arity whose values are sets of tuples of the corresponding arity. For example, a binary relation could be used to represent users' social security numbers, declared as follows: `(defrelation SSN :types (user integer))` The assertion that a user `'U1` has social security number `123456789` is done by: `(++ SSN 'U1 123456789)` and similarly, retraction by: `(-- SSN 'U1 123456789)`

Derived relations — relations whose values are defined by a computation that depends upon the values of other (less derived) relations. (In contrast, non-derived relations have their values explicitly asserted and retracted, as shown for the `SSN` relation above.) For example, the following defines a derived relation that holds of all users whose social security numbers are greater than 500000000 (this may not be a particularly useful concept in the real world!):

```
(DEFRELATION HAS-BIG-SOCIAL-SECURITY-NUMBER
  :DEFINITION
  ((U) S.T. ( > (SSN U =) 500000000)))
```

In the above, `(SSN U =)` is concise syntax for retrieval of the value related to `U` by the `SSN` relation.

Constraints — conditions (expressed in an extension of first order logic), that must be true in every state of the database. If a transition is attempted that would result in a state in which one or more of these conditions is not met, that whole transition is prevented from occurring, and raises an exception.
For example, we can use a constraint to capture the uniqueness of social security numbers, by requiring that no user is related to more than one integer by this relation, expressed as follows:

```
(NEVERPERMITTED
  MULTIPLE-SOCIAL-SECURITY-NUMBERS
   (E (U N1 N2)
      (AND (SSN U N1)
           (SSN U N2)
           (NOT (= N1 N2)))))
```

`MULTIPLE-SOCIAL-SECURITY-NUMBERS` is the name of the constraint.
`(E (P N1 N2) (AND ...))` is the condition, a predicate, taking the form of an existentially quantification ('E' for existential) whose bound variables are `U`, `N1` and `N2`, and whose body is the conjunction `(AND ...)`, capturing the case in which the same user `U` is related to two *different* (`(NOT (= N1 N2))`) integers by the `SSN` relation (`(SSN U N1)` and `(SSN U N2)`).

Constraint repairs — a constraint may be augmented with a 'repair', namely a response to an attempted transition violating the constraint. The purpose is to (try to) automatically recover from such a transition by running the response, which adjusts the transition. The response takes the form of a computation of additional actions to do within the same transition. The union of the actions of the attempted transition and the actions computed by the response, form a new transition, which is then attempted in place of the original transition. This process is iterative, insofar as the new transition (augmented by the suggested additions), may also lead to a violation of some constraint(s), leading to further repairs being activated .... This process either terminates in a transition that leads to a valid state (one in which all constraints are satisfied), or hits a pre-set upper limit on the number of iterations, and raises an exception.
For example, we may augment the social security constraint with a repair that removes the old value of the social security number if a new value is asserted, thus:

```
(NEVERPERMITTED
  MULTIPLE-SOCIAL-SECURITY-NUMBERS
   (E (P N1 N2)
      (AND (SSN P N1)
           (SSN P N2)
           (NOT (= N1 N2))))
   :REPAIR (LAMBDA (P N1 N2)
             (if (PREVIOUSLY (SSN P N1))
                 (-- SSN P N1))))
```

Automation rules — these trigger activity in response to conditions arising. An automation rule comprises a 'trigger' (a predicate) and a 'response' (a statement). Whenever a valid transition occurs, the trigger of each automation rule is evaluated; for those automation rules whose triggers evaluate to true, their responses are then executed. Note the difference between constraint repairs and automation rules: the former react to attempted transitions that violate constraints, whereas the latter react to successfully completed, valid, transitions.
For example, we may define an automation rule to print out the identity of a user whose social security number has changed, as follows:

```
(DEFAUTOMATION NEW-SSN
  ((U N) S.T. (START (SSN U N)))
  (LAMBDA (U N) (PRIN1 U)))
```

`NEW-SSN` is the name of the automation rule,
`((U N) S.T. (START (SSN U N)))` is the condition the rule is watching for (implicitly, an existential quantification over variables `U` and `N` such that...), and
`(LAMBDA (U N) (PRIN1 U))` is the code to be executed in response to the condition being true — the lambda variables get bound to the objects for which the condition is true.

In discussing our implementation's translation of bounded obligations (section 5) into AP5, we will use sans serif font to highlight the use of these AP5 concepts.

4 Illustrative Example - the Library System

As running illustration, we will use examples drawn from the domain of a simple lending library of books. As far as possible, we have striven to use the same examples as in [12] — we will indicate all points of divergence. All the sections of code in `fixed width font` are taken from our our running implementation, and have been changed only to the extent of re-formatting to improve the textual appearance.

Our AP5 representation of the basics of the library system is as shown next (explanation follows immediately afterwards):

```
(DEFRELATION BOOK :DERIVATION BASETYPE)
(DEFRELATION USERORLIBRARY :DERIVATION BASETYPE)
(DEFRELATION USER :DERIVATION BASETYPE
                  :TYPES (USERORLIBRARY))
(DEFRELATION LIBRARY :DERIVATION BASETYPE
                  :TYPES (USERORLIBRARY))

(DEFRELATION HAS :TYPES (BOOK USERORLIBRARY))
(DEFRELATION TODAY :TYPES (INTEGER))
```

The first four forms declare unary relations to serve as types.
The next declaration is of binary relation HAS, which holds between an object of type BOOK and an object of type USERORLIBRARY, modelling who has the book.
The last declaration is of a unary relation TODAY, which holds of an object of type INTEGER, crudely modelling which day it is.

The activity of borrowing a book is defined as follows:

```
(DEFUN BORROW (USER BOOK)
  (IF (?? NOT (HAS BOOK 'LIBRARY))
      (BREAK (FORMAT NIL "BOOK: ~A IS NOT IN THE
          LIBRARY, SO CANNOT BE BORROWED" BOOK)))
  (ATOMIC
   (++ HAS BOOK USER)
   (-- HAS BOOK 'LIBRARY)))
```

The function first checks to ensure the library has the book, and if so, simultaneously retracts that the library has the book (-- HAS BOOK 'LIBRARY), and asserts that the user has the book (++ HAS BOOK USER). It is by enclosing these two statements within the ATOMIC statement that causes them to be done 'simultaneously' in a single database transaction.

5 Our Implementation of Bounded Obligations

5.1 Strict, retractable obligations

A bounded obligation that predicate P must become true before the next state in which Q becomes true is implemented as a pair of AP5 constraints and an a relation to model whether or not the obligation is in effect:

- The first constraint simply prohibits any state in which Q holds and the obligation is in effect.
- The second constraint prohibits any state in which P holds and the obligation remains in effect, but has an associated repair that removes the obligation in such an instance. This repair has the effect of automatically removing the obligation once P has been achieved, so that if in some later state Q becomes true, the first constraint will not be violated, because the obligation, having already been met, no longer holds.

Note that this implementation strictly prohibits *any* violation of the obligation.

Our implementation allows the programmer to declare an obligation in a straightforward manner. It automatically generates the above two corresponding AP5 constraints. This is far less burdensome on the programmer than requiring their explicit definition.

A Strict Obligation of the Library Example: The first obligation of the library example that we consider is: *a user must return a book within 21 days of borrowing it.* As in [12], we introduce the concept of a 'due date', and express the obligation in terms of this. Binary relation BOOKDUE is defined to hold between a book and the date (if any) by which that book must be returned to the library:

```
(DEFRELATION BOOKDUE :TYPES (BOOK INTEGER))
```

An obligation that requires the book to be back in the library at or before the due date is declared as follows:

```
(OBLIGE   "return books"
          "(U B)"
          "(HAS B 'LIBRARY))"
          "(> (TODAY =) (BOOKDUE B =))")
```

The first argument,[1] `return books`, is the name of the obligation. The second argument, `(U B)`, is a list of variables over which the obligation is to be existentially quantified (details will become obvious shortly). The third argument, `(HAS B 'LIBRARY))`, is the predicate that must be satisfied to meet the obligation. The fourth argument, `(> (TODAY =) (BOOKDUE B =))`, is the

[1] Each argument takes the form of a Lisp string, hence the surrounding string quotes

predicate that defines the state before which the obligation must have been met, namely when today's date has passed the date on which the fine was due.

Our system generates the following implementation from the above declaration:

```
(NEVERPERMITTED
  |return books obliged-but-not-done|
   (E (U B)
      (AND (OBLIGATION2 "return books" U B)
           (NOT (HAS B 'LIBRARY))
           (> (TODAY =)
              (BOOKDUE B =)))))

(NEVERPERMITTED
  |return books achieved-and-still-obliged|
   (E (U B)
      (AND (OBLIGATION2 "return books" U B)
           (HAS B 'LIBRARY)))
   :REPAIR
    (LAMBDA (U B)
      (-- OBLIGATION2 "return books" U B)))
```

The first constraint, called |return books obliged-but-not-done|, prohibits any state in which its predicate, (E (U B) (AND ...)), is true, namely, in which there exists objects U and B for which:
— the return books obligation[2] holds,
(OBLIGATION2 "return books" U B),
— the library does not have B,
(NOT (HAS B 'LIBRARY)), and
— today's date is past the due date of B,
(> (TODAY =) (BOOKDUE B =)).

The second constraint, called |return books achieved-and-still-obliged|, prohibits the obligation from holding between U and B if the library has B; associated with it is the repair that retracts the obligation (by means of the statement (-- OBLIGATION2 "return books" U B)).

To make use of this obligation, the programmer must extend the definition of borrow to insert the appropriate due date, and to insert that the obligation to return the book holds, as shown in the last two lines added in the definition of the function borrow:

```
(DEFUN BORROW (USER BOOK)
  (IF (?? NOT (HAS BOOK 'LIBRARY))
      (BREAK (FORMAT NIL "BOOK: ~A IS NOT IN THE
          LIBRARY, SO CANNOT BE BORROWED" BOOK)))
  (ATOMIC
   (++ HAS BOOK USER)
   (-- HAS BOOK 'LIBRARY)
   (UPDATE BOOKDUE OF BOOK TO (+ (TODAY ?) 21))
   (++ OBLIGATION2 "return books" USER BOOK)))
```

[2] (OBLIGATIONi o $a_1 \ldots a_i$) for integer i is used to relate the obligation named o to the objects $a_1 \ldots a_i$.

UPDATE is a syntactic a shorthand for simultaneously retracting the old value and asserting the new value.

Note that our strategy is to have a single rule quantified over users and books, and to use the OBLIGATION2 relation to turn on and off obligations for a given user and book (turning off an obligation is the means by which it can be retracted). This matches the style in [12]. An alternative would have been to instantiate a separate rule for each particular user and book for which the obligation was to hold, asserting and de-asserting the rule in order to turn on and off the obligation.

5.2 Violatable obligations

To render an obligation that predicate P must become true before the next state in which Q becomes true violatable', that is, to allow behaviors in which the obligation is *not* met, the AP5 implementation is adjusted as follows:

- The constraint that prohibits any state in which Q holds and the obligation remains in effect is removed.
- A derived relation is introduced, defined to hold in those and only those states in which Q holds and the obligation is in effect.

The net effect of these two changes is to now permit behaviors which violate the obligation, but the derived relation introduced will automatically hold in states in which the violation occurs. This derived relation matches the 'normative' predicate introduced in [12].

Interestingly, this step corresponds exactly to the weakening of constraints presented in [2]. Balzer used it to allow databases to include instances of 'exceptional' data, that is, data not meeting integrity constraints weakened in this manner. Our application is a special case of this, where we weaken specifically the constraint introduced to require adherence to a bounded obligation.

Our implementation allows the programmer to declare an obligation to be 'exceptionalized'. It automatically retracts the constraint rule that enforced adherence to the constraint, and adds the derived relation recognizing violations.

Library example:
Conversion of the strict obligation to one which can be violated, accompanied by a derived relation defined to hold when the violation occurs, is achieved by issuing the following declaration:

```
(EXCEPTIONALIZE-OBLIGATION "return books")
```

Recall that return books was the name given the obligation in its original declaration. The effect of

this is to remove the constraint named |return books obliged-but-not-done|, and introduce a derived relation of the same name, defined as follows:

```
(DEFRELATION
 |return books obliged-but-not-done|
  :DEFINITION
  ((U B) S.T.
   (AND (OBLIGATION2 "return books" U B)
        (NOT (HAS B 'LIBRARY))
        (> (TODAY =)
           (BOOKDUE B =)))))
```

5.3 Response to violation of obligations

Typically, the system being specified should react in some manner to the occurrence of obligation violations. Our implementation defines such a reaction in terms of an AP5 automation rule, whose trigger is the value of the derived relation that monitors for such violations becoming true, and whose response is whatever code the specifier wishes to have executed.

Library example:
If a user fails to return a book before the due date, then a fine will be issued and the due date on the book will be extended by seven days.

Introduction of this response[3] to failure to return a library book is achieved by the following declaration:

```
(RESPOND-TO-UNMET-OBLIGATION
 "return books"
 "(ATOMIC
   (UPDATE USERBOOKFINE OF U B TO
    (+ (THEONLY F S.T. (USERBOOKFINE U B F)
        :IFNONE 0) 5))
   (UPDATE BOOKDUE OF B TO
    (+ (BOOKDUE B ?) 7)))")
```

Again, return books is the name originally given to the obligation; this causes the generation of an automation rule that responds to the violation of the obligation (detected by examining values of the derived relation introduced earlier) by executing the provided code (ATOMIC (UPDATE ...)):

```
(DEFAUTOMATION |return books unmet-response|
  ((U B) S.T.
   (START (|return books obliged-but-not-done|
           U B)))
  (LAMBDA (U B)
   (ATOMIC
     (UPDATE USERBOOKFINE OF U B TO
      (+ (THEONLY F S.T. (USERBOOKFINE U B F)
          :IFNONE 0) 5))
     (UPDATE BOOKDUE OF B
                      TO(+ (BOOKDUE B ?) 7)))))
```

[3] We have arbitrarily chosen the amount of fine to be 5.

Having done this, we can now experiment with our library specification, observing that as a user fails to return a book on time a fine is levied, and the due date on the book is extended.

In fact, there is more to the treatment of fines: *The fine must be paid within seven days of issue.* Note that this is another *bounded obligation*. To introduce this, we proceed in a manner similar to the way in which we introduced the obligation to return books by their due dates. We declare a ternary relation finedue to hold between a user, a book and a day (in our library system, it is possible for a user to accrue fines for failing to have returned several books, at various different times; it is also possible for a book to have been tardily returned by more than one user, giving rise to fines for each of those users. Hence the need to keep track of fines due not just per user or per book, but per user and book): (DEFRELATION FINEDUE :TYPES (USER BOOK INTEGER))
We then declare the following obligation:

```
(OBLIGE "pay fines"
        "(U B)"
        "(NOT (> (USERBOOKFINE U B =) 0))"
        "(> (TODAY =) (FINEDUE U B =))")
```

As with the declaration of the obligation for returning books, its four arguments are the name, pay fines, by which to refer to the obligation, a list of variables, (U B), over which the obligation will be existentially quantified, the predicate that is obliged to be achieved, (NOT (> (USERBOOKFINE U B =) 0), namely that the amount of fine not be greater than zero, and the predicate that defines the state by which the obligation must be met, (> (TODAY =) (FINEDUE U B =)), namely, when today's date has passed the date on which the fine was due.

We re-declare our response to failing to return a book to not only levy a fine, but also set the due date of that fine, and establish the obligation to pay it:

```
(RESPOND-TO-UNMET-OBLIGATION
 "return books"
 "(ATOMIC
   (UPDATE USERBOOKFINE OF U B TO
    (+ (THEONLY F S.T. (USERBOOKFINE U B F)
        :IFNONE 0) 5))
   (UPDATE BOOKDUE OF B TO
    (+ (BOOKDUE B ?) 7)))")
   (IF (?? NOT (> (USERBOOKFINE U B =) 0))
       (UPDATE FINEDUE OF U B TO
        (+ (TODAY ?) 7)))
   (++ OBLIGATION2 \"pay fines\" U B))")
```

Re-declaration replaces the old AP5 code with the translation of the above.

A subtle case is one in which the user has already accumulated a fine for not having returned that book

(possible if the user borrowed the book, returned it *after* its due date [thus incurring a fine], re-borrows the same book, and now is late returning it a second time!); our response recognizes this case by the existence of a non-zero fine levied against the user for that book, and if so, increases the fine and leaves unchanged the already established fine due date.[4]

The obligation to pay a fine can be violated. The library specification reflects this possibility, and specifies the following response:
If a fine is not paid on time, then the borrowing rights of the user are blocked, the fine increased, and the due date on the fine extended by another 7 days.[5]
Our declarations to effect the above are as follows:

```
(EXCEPTIONALIZE-OBLIGATION "pay fines")
```

```
(RESPOND-TO-UNMET-OBLIGATION
   "pay fines"
   "(ATOMIC
     (++ BLOCKED-BORROWING-RIGHTS U)
     (UPDATE USERBOOKFINE U B TO
      (+ (THEONLY F S.T. (USERBOOKFINE U B F)
          :IFNONE 0) 5))
     (UPDATE FINEDUE OF U B TO
      (+ (TODAY ?) 7)))")
```

Note again that the declaration need provide only the name given to the original obligation, `pay fines`, and the response code to be run when this obligation is violated (`ATOMIC ...`). Our implementation's translation of this into the equivalent AP5 constraint repair is similar to the earlier treatment of the response to `return books`, so we omit the result of this translation here. To make use of this we manually augment the definition of borrow to require that a user's borrowing rights not be blocked.

5.4 A Library Example Scenario

The following is a scenario that we have run through the version of our library example with both the obligations mentioned above (`return books` and `pay fines`) declared along with their responses.

In the initial state, no user's borrowing rights are blocked, all the books are in the library, there are no books due, today is day 1, there are no outstanding obligations, no fines, and no due dates for fines. In this and the successive stages, we give a simple printout of this information in the following form:

```
BLOCKED USERS: NIL     OBLIGATIONS: NIL
HAS: NIL               FINES: NIL
DUE: NIL               FINESDUE: NIL
TODAY: 1
```

Step 1: user `'U1` borrows book `'B1`: `(BORROW 'U1 'B1)`, resulting in the state:

```
BLOCKED USERS: NIL     OBLIGATIONS:
HAS: ((B1 U1))          ((return books U1 B1))
DUE: ((B1 22))         FINES: NIL
TODAY: 1               FINESDUE: NIL
```

Step 2: The date advances to 23: `(UPDATE TODAY OF TO 23)`, resulting in the user `U1` being fined for not having returned book `'B1` on time:

```
BLOCKED USERS: NIL     OBLIGATIONS:
HAS: ((B1 U1))          ((pay fines U1 B1)
DUE: ((B1 29))           (return books U1 B1))
TODAY: 23              FINES: ((U1 B1 5))
                       FINESDUE: ((U1 B1 30))
```

Step 3: The user `'U1` returns the book `'B1`: `(RETURNBOOK 'U1 'B1)`, resulting in the removal of the obligation to return the book (the due date sticks around, but is of no further consequence, and would be reset the next time the book gets borrowed), but retention of the obligation to pay the fine, of course:

```
BLOCKED USERS: NIL     OBLIGATIONS:
HAS: NIL                ((pay fines U1 B1))
DUE: ((B1 29))         FINES: ((U1 B1 5))
TODAY: 23              FINESDUE: ((U1 B1 30))
```

Step 4: The date advances to 31: `(UPDATE TODAY OF TO 31))`, resulting in the increment of the fine, the extension of the fine due date, and the blocking of user `U1`'s borrowing rights:

```
BLOCKED USERS: (U1)    OBLIGATIONS:
HAS: NIL                ((pay fines U1 B1))
DUE: ((B1 29))         FINES: ((U1 B1 10))
TODAY: 31              FINESDUE: ((U1 B1 38))
```

6 Related Work and Conclusions

As mentioned earlier, [12] provides the description of bounded obligations which our implementation emulates. The only significant difference is that their formalism rests upon a deontic *action* logic, and thus refers to the events that occur in the transitions *between* states, whereas our implementation is state-based, and refers to the contents of the states themselves.

Of the forms of obligations discussed in [13], our implementation provides the capabilities of their **requiring to have ... by ... or-else ...** construct, plus

[4] In [12], failure to return the book the second time around appears *not* to cause the levying of any additional fine.

[5] In [12], the specified response *also* extended the due date of the book, regardless of whether or not that book had been returned and possibly re-borrowed by some other user.

permits the retraction of such obligations. The AP5 system upon which our implementation is based is akin to what they call a system that handles 'transitional violation' by means of 'an abstraction which makes improper states invisible to the rest of the system'. Whereas they show a 'transaction'-like mechanism expressed in terms of their obligation construct, our implementation operates in the opposite direction, demonstrating an obligation construct in terms of a 'transaction' mechanism.

We feel that to implement bounded obligations with comparable ease on some other language base, the key abilities we would need are those of:
— recognizing when a transaction violates arbitrary integrity constraints,
— the option of running activities that (attempt) to 'repair' the transaction to make it lead to a consistent state, and
— within those 'repairs', the ability to refer to the details of transaction itself, as well as the consistent state prior immediately prior to the attempted transaction.
See [10] for a more general discussion of the relationships between these kind of capabilities.

Other members of our group have explored the virtues of building upon the AP5 constraint mechanisms; in [8] these mechanisms lie to the heart of a support environment for programming; in [14] cooperative software development is facilitated by allowing *proposed* changes to be broadcast, so that where those changes would clash with system-wide constraints, they can be cooperatively adjusted prior to actually making those changes, thus permitting, as the authors put it, a 'lazy' style of consistency management. As mentioned in section 5.2, Balzer has followed a different approach to coping with changes that would violate a constraint, namely adjusting the constraint to permit exceptions, and marking those exceptions accordingly [2]. This latter is the one that corresponds to our treatment of bounded obligations. Generally, we find the AP5 environment to be conducive to investigations of a variety of software research issues.

The idea of permitting violatable constraints has also been studied in [3], wherein a properties of, and algorithms for, *constraint hierarchies* are presented. Briefly, these permit the statement of *preferential* constraints, which the system must try to satisfy, but is not required to do so. A very similar approach is to be found in [9], wherein constraints are organized into a *relaxation lattice*. Our naive impression of how the requirements of the library example might be expressed in this style are along the lines of:

Most preferred: A user returns a library book by its due date.
Next most preferred: The user who has failed to return a library book by one or more of its due dates will be fined an amount equal to the standard fine multiplied by the number of due dates missed.
Most preferred: A user will pay a fine by its due date, etc.
The impression we have is that constraint hierarchies provide a powerful, general purpose mechanism, but as with AP5's constraint mechanisms, there is some distance between the natural statement of bounded-obligation style requirements and their encoding in terms of these mechanisms.

Incremental software development, in which an overly ideal but easy to comprehend version of some system is developed first, and thereafter incrementally elaborated to deal with the complexities of the exceptional cases, is also a continuing theme of investigation both within our group and in the broader community. Our group has developed a library of so-called 'evolution transformations' [1, 6, 11], which are applied to create the next version of a specification by modifying the semantics of the current version. Our translators for the declarations of 'exceptionalize' and 'response' to bounded obligations could be regarded as simple such evolution transformations.

To conclude, we speculate that bounded obligations could be linked to even higher-level goals of system design. For example, the *purpose* of levying fines against users is to encourage users to return books on time (which in turn is to facilitate the fair distribution of limited resources — books — among the library users). Likewise, once fines have been levied, users need to be encouraged to pay them (for otherwise they would be no deterrent to the tardy return of borrowed books), hence the ultimate threat of removing a user's borrowing rights. Thus we see that bounded obligations, and the responses if they are violated, are themselves implementations of some higher-level objectives. We think it would be interesting to explore their inclusion within the higher-level considerations of requirements and design such as have been reported in [7, 5]. Seen in this light, the content of bounded obligations are not arbitrary; rather, there are canonical forms of exceptions to constraints, and canonical responses to such exceptions (that both discourage the occurrence of those exceptions, and help restore the preferred state of the system). It would be poor design to impose an obligation on an agent who does not have the capability to meet that obligation (e.g., impose the obligation on a library user to ensure

that another, unrelated, user returns a book). Similarly, it would be poor design to impose an obligation on an agent and also give the agent the capability to *retract* the obligation (e.g., impose the obligation on a library user to return a book and also give that user the capability to retract that obligation - presumably only the library staff should have such a capability). This is an area we feel is deserving of further investigation.

Acknowledgements

Bob Balzer's Software Sciences Division here at ISI, of which I am a member, has provided the research context and foundation for this work. Particular thanks are due to Don Cohen, one of the implementors of AP5, for his advice on its use.

References

[1] R. Balzer. Automated enhancement of knowledge representations. In A. Joshi, editor, *Proceedings, 9th International Joint Conference on Artificial Intelligence, Los Angeles*, pages 203–207, August 1985.

[2] R. Balzer. Tolerating inconsistency. In *Proceedings, 13th International Conference on Software Engineering, Austin, Texas, USA*, pages 158–165. IEEE Computer Society Press, August 1991.

[3] A. Borning, B. Freeman-Benson, and M. Wilson. Constraint hierarchies. *Lisp and Symbolic Computation*, 5:223–270, 1992.

[4] D. Cohen. Compiling complex database transition triggers. In *Proceedings, ACM SIGMOD International Conference on the Management of Data, Portland, Oregon*, pages 225–234. ACM Press, 1989. SIGMOD RECORD Volume 18, Number 2, June 1989.

[5] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal-directed requirements acquisition. *Science of Computer Programming*, 20(1-2):3–50, April 1993.

[6] M.S. Feather. Constructing specifications by combining parallel elaborations. *IEEE Transactions on Software Engineering*, 15(2):198–208, February 1989.

[7] S. Fickas and P. Nagarajan. Critiquing software specifications. *IEEE Software*, pages 37–47, November 1988.

[8] N. Goldman and K. Narayanaswamy. Software evolution through iterative prototyping. In *Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia*, 1992.

[9] M.P. Herlihy and J.M. Wing. Specifying graceful degredation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, January 1991.

[10] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 455–467, 1991.

[11] W.L. Johnson and M.S. Feather. Building an evolution transformation library. In *Proceedings, 12th International Conference on Software Engineering, Nice, France*, pages 238–248. IEEE Computer Society Press, March 1990.

[12] S.J.H. Kent, T.S.E. Maibaum, and W.J. Quirk. Formally specifying temporal constraints and error recovery. In *Proceedings of the IEEE International Symposium on Requirements Engineering, San Diego, CA, USA, January 1993*, pages 208–215. IEEE Computer Society Press, 1993.

[13] N.H. Minsky and A.D. Lockman. Ensuring integrity by adding obligations to priveleges. In *Proceedings, 8th International Conference on Software Engineering*, pages 92–102. IEEE Computer Society Press, 1985.

[14] K. Narayanaswamy and N. Goldman. "Lazy" Consistency: A Basis for Cooperative Software Development. In *Proceedings of the Conference on Computer Supported Cooperative Work, Toronto, Canada*, pages 257–264. ACM Press, 1992.

[15] D.E. Perry. Version control in the inscape environment. In *Proceedings, 9th International Conference on Software Engineering, Austin, Texas, USA*, pages 142–149. IEEE Computer Society Press, 1987.